Correction : Examen partiel

Exercice 1 Un code concis est:

```
public static double TriPasc(int n){
    double[ ][ ] mat = new double [n+1][ ];
    for(int i = 0 ; i < n+1 ; i++) {
      mat[i] = new double[i+1];
      mat[i][0] = mat[i][i] = 1;
      for(int j = 1 ; j <= i-1 ; j++)
         mat[i][j] = mat[i-1][j-1] + mat[i-1][j];
   }
   return mat[n][n/2];                                  // return 1.0;   ou rien    ok
}
```

Exercice 2

1. Si $p \leq n$, la fonction récursive

```
public static double xx(int n, int p){
    if(p==n || p==0)
      return 1.0;
    else
      return xx(n-1,p-1)+xx(n-1,p);
  }
```

renvoie $\binom{n}{p}$, le nombre de sous-ensembles de $[n] := \{1, \ldots, n\}$ ayant p éléments. Sinon, elle ne s'arrête jamais.

2. Si $p > n$, `xx(n-1,p)` n'atteint jamais les conditions limites $p = n$ ou $p = 0$, d'où le fait que l'algorithme ne s'arrête pas. Si $p \leq n$, montrons par récurrence sur n que (1.) est bien vrai pour $p = 0, \ldots, n$. Pour $n = 0$ et pour $p = 0$ ou n c'est bien vrai car `xx(n,p)` retourne 1 (soit $p = n$ et c'est vrai car seul $[n]$ tout entier a $p = n$ éléments, soit $p = 0$ et c'est vrai car seul $\emptyset$ a $p = 0$ éléments). Pour $n > 0$ et $0 < p < n$, par récurrence, `xx(n-1,p-1)` et `xx(n-1,p)` sont valides. Puisque $[n]$ a $\binom{n-1}{p-1}$ sous-ensembles de taille p contenant l'élément 1, et $\binom{n-1}{p}$ sous-ensembles de taille p ne contenant pas l'élément 1, on en déduit que `xx(n,p)` est valide.

3. La complexité de

```
public static double yy(int n){
    for(int i=0; i<= n ; i++)
              xx(n,i)
  }
```

est $\Theta(2^n)$ car le nombre de `return 1.0` effectués par un appel à `xx(n,i)` est $\binom{n}{i}$, et le nombre total est $\sum_{i=0}^{i=n} \binom{n}{i}$, qui correspond au nombre total de sous-ensembles de $[n]$, soit 2^n.

Exercice 3

1. Pour $p = 0$, on a $n = 1$ et $T(n) = 1 = O(1)$. $T(b^{p+1}) = aT(b^p) + f(b^{p+1}) = a\big(a^p + \sum_{i=0}^{i=p-1} a^i f(b^{p-i})\big) + f(b^{p+1}) = a^{p+1} + \sum_{i=0}^{i=p-1} a^{i+1} f(b^{p-i}) + f(b^{p+1}) = a^{p+1} + \sum_{i=0}^{i=p} a^i f(b^{p+1-i})$.

2. On a $\log_b n = \log_b a \log_a n$, car avec $n = b^p$, $a = b^x$ et $n = a^y$, alors $b^p = n = b^{xy}$, d'où $p = xy$. D'où $a^p = a^{\log_b n} = a^{\log_b a \log_a n} = \left(a^{\log_a n}\right)^{\log_b a} = n^{\log_b a}$.
3. (a) Pour $a = 8$, car on a $n^2 = O(n^{\log_2 a - \epsilon})$, d'où $T(n) = \Theta(n^{\log_2 a})$.
   (b) Pour $a < 4$, car on a $n^2 = \Omega(n^{\log_2 a + \epsilon})$, d'où $T(n) = \Theta(f(n))$.
   (c) Jamais.

Exercice 4

1. On effectue au moins n passages dans la boucle `for` d'où $\Omega(n)$. On effectue au plus n passages dans la boucle `while` d'où $O(n^2)$.
2. La complexité $T(n)$ est proportionnelle au nombre de passages dans la boucle `while`. En notant $b_i(j)$ le nombre de fois qu'on a effectué l'instruction `Tab[j]=false` ou `true` au $i$ème passage dans la boucle `for`, on a $T(n) = \sum_{j=0}^{j=n-1} b_n(j)$. De plus, $b_i(j)$ satisfait les conditions initiales et de récurrence décrites, donc $b_n(j) = \lfloor \frac{n}{2^j} \rfloor$. D'où $T(n) = \sum_{j=0}^{j=n-1} \lfloor \frac{n}{2^j} \rfloor \leq n \sum_{j \geq 0} \frac{1}{2^j} \leq 2n$. D'où $T(n) = O(n^2)$.

Exercice 5

1. Code java:

```
static int Alkwa (int a, int b){
  int c=0;
  while(a !=0){c+=(a%2)*b; a/=2; b*=2;}
  return c;
}
```

2. Soient a et b les valeurs de `a` et `b` données en paramètre d'entrée.
   On a $a = \sum_{i=0}^{i=n} a_i 2^i$ avec $a_i \in \{0,1\}$ pour tout $i = 0, \ldots, n$. Après un nombre k de passages dans la boucle while, on a les trois invariants suivants:
   (a) `a` $= \sum_{i=k}^{i=n} a_i 2^{i-k}$
   (b) `b` $= b2^k$
   (c) `c` $= \sum_{i=0}^{i=k-1} a_i b 2^i$

   En effet, pour $k = 0$ c'est clairement vrai puisque `a` $= a$, `b` $= b$, et `c` $= 0$. Puis l'invariant 1 se conserve car faire `a/=2` revient à faire $2^{i-k} \leftarrow 2^{i-k-1}$ et $a_k \leftarrow 0$ dans l'égalité (a). L'invariant 2 se conserve clairement par `b*=2`. L'invariant 3 se conserve par l'instruction

   `c+=(a%2)*b`

   car, par (a), on a `a%2`$= a_k$, et par (b), on a `b`$= b2^k$.
   Donc l'algorithme retourne `c`$= \sum_{i=0}^{i=n} a_i b 2^i = ab$.